

uniqname (REQUIRED)

EECS 280 Final Exam

Fall 2017

This is a closed-book exam. You may use one note sheet, 8.5"x11", double-sided, with your name on it. This exam has 5 problems on 16 pages, and it is out of 80 points total. In addition to this exam booklet, you are also provided a reference packet with scratch space.

Read the entire exam through before you begin working. Work on those problems you find easiest first. Read each question carefully, and note all that is required of you. Keep your answers clear and concise, and state all of your assumptions carefully. Write your answers in the space provided. Assume all code is in standard C++11, and use only standard C++11 in your solutions.

Write your uniqname on the line provided at the top of each page.

You are to abide by the University of Michigan/Engineering honor code. To receive a grade, sign below to signify that you have kept the honor code pledge:

I have neither given nor received aid on this exam, nor have I concealed any violations of the Honor Code.

Signature:	<hr/> <hr/>
Name:	<hr/> <hr/>
Uniqname:	<hr/> <hr/>
UMID:	<hr/> <hr/>
Exam Room Number:	<hr/> <hr/>

First and last name of person sitting to your **right** (write the symbol \perp if at the end of a row):

First and last name of person sitting to your **left** (write the symbol \perp if at the end of a row):

Correctly completing this cover page is worth **1 point**.

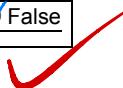
EECS 280 Fall 2017 Final 1/16
uniqname (REQUIRED)

Problem 1: Short Answer (19 Points)

1a) (6 points) For each of the following, fill in the bubble next to **True** or **False**. (Make sure to erase completely if you change your answer.)

i) The Rule of the Big Three states that if a class has member variables that are pointers, it must implement custom versions of the copy constructor, assignment operator, and destructor.

<input type="radio"/> True	<input checked="" type="radio"/> False
----------------------------	--



ii) It is legal for a pointer whose type is `int *` to contain the address of an `int` object on the stack.

<input checked="" type="radio"/> True	<input type="radio"/> False
---------------------------------------	-----------------------------



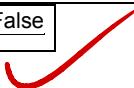
iii) If class A contains a declaration of the form `friend class B;`, then the code in class A can access the private members of class B.

<input type="radio"/> True	<input checked="" type="radio"/> False
----------------------------	--



iv) A try block cannot have more than one catch block that is associated with it.

<input type="radio"/> True	<input checked="" type="radio"/> False
----------------------------	--



v) If a recursive call is on the last line of a function, then the function is always tail recursive.

<input type="radio"/> True	<input checked="" type="radio"/> False
----------------------------	--



vi) A person suffering from impostor syndrome does not recognize their own accomplishments and believes they do not truly belong in the field in which they work.

<input checked="" type="radio"/> True	<input type="radio"/> False
---------------------------------------	-----------------------------



EECS 280 Fall 2017 Final 2/16

unique name (REQUIRED) _____

1b) (6 points) Consider the following class definition.

```
class RobotArm {  
public:  
    RobotArm() : num_joints(0) {}  
    void add_joint(const std::string &name) {  
        joints[num_joints] = new Joint { name };
```

```

    ++num_joints;
}

void remove_joint() {
    if (num_joints > 0) {
        --num_joints;
        delete joints[num_joints];
    }
}

private:
    struct Joint {
        std::string name;
    };
    Joint * joints[10];
    int num_joints;
};

```

For each of the lines of code below, determine if memory is leaked when the lines execute.

Memory leak when the lines below execute?	Yes	No
int main() { // program starts	<input type="radio"/>	<input checked="" type="radio"/>
RobotArm *ra = new RobotArm;	<input type="radio"/>	<input checked="" type="radio"/>
ra->add_joint("wrist");	<input type="radio"/>	<input checked="" type="radio"/>
ra->add_joint("elbow");	<input type="radio"/>	<input checked="" type="radio"/>
ra->remove_joint();	<input type="radio"/>	<input checked="" type="radio"/>
delete ra;	<input checked="" type="radio"/>	<input type="radio"/>
<i>This is what causes the program to lose track of the "wrist" joint</i>	<input checked="" type="radio"/>	<input type="radio"/>
return 0; // program exits	<input type="radio"/>	<input checked="" type="radio"/>

错因: remove_joint() 一次只会 delete 一个 joint, 而不会像 clean() 全部删干净。

EECS 280 Fall 2017 Final 3/16

uniqname (REQUIRED) _____

1c) (4 points) Consider the following function template and class definition.

```

template <typename T>
bool less_than(T x, T y) {
    return x < y;
}

```

```

class Duck {
    std::string name;
public:
    Duck(std::string name_in)
        : name(name_in) {}
};

```

What is the result of each call to `less_than()` below? If the code does not compile, select "Compile Error". If execution of the statement crashes when it is run, select "Runtime Error". If the code compiles and the call to `less_than()` returns true, select "True", and if it returns false, select "False".

Call to <code>less ()</code>	Compile Error	Runtime Error	True	False
<code>less_than(5, 10);</code>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
<code>std::string s1 = "hello";</code> <code>std::string s2 = "world";</code> <code>less_than(s1, s2);</code>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>
<code>Duck duck1("Huey");</code> <code>Duck duck2("Dewey");</code> <code>less_than(duck1, duck2);</code>	<input checked="" type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
<code>Duck ducks[1] = { Duck("Louie") };</code> <code>less_than(ducks, ducks + 1);</code>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input checked="" type="radio"/>

EECS 280 Fall 2017 Final 4/16

unique name (REQUIRED)

1d) (3 points) Consider the following program.

```
#include <iostream>
class Exception {};
class DivisionException {};
class NotImplementedException {};

double reciprocal(int x) {
    if (x == 0) { throw DivisionException(); }
    return 1.0 / x; [0.5] [0.1]
}

double read_and_reciprocate() {
    int input;
    try {
        if (std::cin >> input) {
            return reciprocal(input);
        } else {
            throw Exception();
        }
    } catch (NotImplementedException &e) {
        std::cout << "Not implemented" << std::endl;
    }
}
```

```

        return 0;
    }

int main() {
    try {
        std::cout << read_and_reciprocate() << std::endl;
        std::cout << read_and_reciprocate() << std::endl;
    } catch (DivisionException &e) {
        std::cout << "Bad division" << std::endl;
    }
}

```

For each of the following inputs on standard input, what does the program print? If nothing is printed, write "nothing". If the program crashes when it is run, write the output that is printed before it crashes, as well as "runtime error".

Input	Output
2	0.5 (runtime error)
2 10	0.5 0.1
0 1	Bad division (runtime error)

only: Bad division

when the DivisionException is caught.

the second read-and-reciprocate() is skipped.

Problem 2: Dynamic Memory (16 Points)

➡ Consult the separate **Problem 2 Reference Sheet** for this problem. ⬅

2a) (5 points) Implement the `Cinema::add_movie` function according to the RME. Your implementation **MUST NOT** leak any memory.

```

// REQUIRES: num_movies <= MAX_MOVIES
// MODIFIES: *this
// EFFECTS: Creates a new movie with the given title and length, // and
// adds the new movie to the end of the movies array. // If the array is
// already full, then the last item will // be removed and replaced by the
// new one.
void Cinema::add_movie(const string& title, int length) {

```

```
void Cinema::add_movie(const string &title, int length){  
    if(num_movies==20){  
        Movie *old_movie = movies[19];  
        delete old_movie;  
        num_movies--;  
    }  
    Movie *new_movie = new Movie(title,length);  
    movies[num_movies] = new_movie;  
    num_movies++;  
}
```

```
}
```

EECS 280 Fall 2017 Final 6/16
unique name (REQUIRED) _____

2b) (7 points) Implement a copy constructor for Cinema. It should make a deep copy of all the movies in the array and MUST use the add_movie function from above. Write it as it would appear outside the class definition.

```
Cinema::Cinema (const Cinema &other):name(other.name),num_movies(other.num_movies){  
    for (int i = 0; i < num_movies; ++i){  
        const string title = other.movies[i]->get_title();  
        int length = other.movies[i]->get_length();  
        this->add_movie(title, length);  
    }  
}
```

Both array movies in heap. Both array with pointer by themselves in heap.

EECS 280 Fall 2017 Final 7/16
unique name (REQUIRED) _____

2c) (4 points) Which of the following **must** be defined to make the following code work successfully (compile and have no memory errors or leaks)? Assume the relevant libraries, header files, and using declarations are included. Fill in the bubble for Yes or No below for each item. (Make sure to erase completely if you change your answer.)

```
int main() {
    Cinema Rave_Cinemas("Rave");
    Rave_Cinemas.add_movie("Justice League", 119);
    Rave_Cinemas.add_movie("Star Wars: The Last Jedi", 150);
    Cinema Quality_16(Rave_Cinemas); deep copy / copy constructor.
    Cinema Michigan_theater = Quality_16;
    copy constructor.
    Movie old_movie("Guess who's coming to dinner", 120);
    Quality_16.add_movie(old_movie);
    cout << old_movie.get_title() << endl;
    cout << Quality_16 << endl;
    return 0;
}
```

Must be defined to avoid errors in the code above?	Yes	No
A default constructor for Cinema	<input type="radio"/>	<input checked="" type="radio"/>
An overloaded assignment operator for Movie	<input type="radio"/>	<input checked="" type="radio"/>
An overloaded assignment operator for Cinema	<input type="radio"/>	<input checked="" type="radio"/>
An overloaded add_movie function that takes in a Movie	<input checked="" type="radio"/>	<input type="radio"/> X
A destructor for Movie	<input type="radio"/>	<input checked="" type="radio"/>

	Yes	No
A destructor for Cinema	<input checked="" type="checkbox"/>	<input type="checkbox"/>
An overloaded <code>operator<<</code> for Movie	<input type="checkbox"/>	<input checked="" type="checkbox"/>
An overloaded <code>operator<<</code> for Cinema	<input checked="" type="checkbox"/>	<input type="checkbox"/>

EECS 280 Fall 2017 Final 8/16

unique name (REQUIRED)

Problem 3: Lists and Templates (14 Points)

Consider the following class template:

```
template <typename T>
class List {
public:
    // EFFECTS: Initializes this list to be empty.
    List() : first(nullptr), last(nullptr) {}

    // YOU WILL WRITE THIS IN 3b) ←
    void truncate(const T &item);

    // Assume the big three are defined as needed.
private:
    struct Node {
        T datum; // the element stored in this Node
        Node *next; // the next Node, or null if there is none Node
        *prev; // the previous Node, or null if there is none ;
        Node *first; // the first Node in the list, or null if it is empty Node
        *last; // the last Node in the list, or null if it is empty
    };
}
```

For 3a) and 3b), you may **NOT** use any functions that do not appear in the class declaration above. Your functions **MUST** be iterative, and you may **NOT** use recursion. Your code must **NOT** leak any memory. You **MAY** assume that the type T supports the == and != operators.

3a) (4 points) Implement `find()` according to its RME.

```
// EFFECTS: Returns a pointer to the first node that contains 'item', // or
nullptr if 'item' is not in the list. Node * find(const T &item) {
```

```

Node *find(const T &item){
    for (Node *i = first; i != nullptr; i = i->next){
        if (i->datum == item){
            return i;
        }
    }
    return nullptr;
}

```

EECS 280 Fall 2017 Final 9/16
 uniqname (REQUIRED) _____

3b) (10 points) Implement `truncate()` according to its RME. You **MUST** use `find()`, and you may assume it is implemented correctly.

```

// MODIFIES: *this
// EFFECTS: If 'item' is in the list, removes the first appearance of // 'item' and all elements that follow from the list. // EXAMPLE: If 'list' contains [2, 3, 1, 4, 1], then list.truncate(1) // modifies 'list' to contain [2, 3].
// If 'list' contains [2, 3, 1, 4, 1], then list.truncate(5) // does not change 'list'.
void truncate(const T &item) {

```

```

    void truncate (const T &item){
        if (find(item)){
            Node *position = find(item);
            position->prev->next = nullptr;
            last = position;
        }
    }
}

```

错因：

①没有考虑到如果目标是第一个的情况。

②没有将废弃的在heap上delete掉。

```

template<typename T>
void truncate (const T &item){
    Node* target = find(item);
    if (target){
        last = target->prev;
        if (last==nullptr){
            first = last = nullptr;
        } else{
            last -> next = nullptr;
        }
    }
    while (target!=nullptr){
        Node *temp = target;
        delete temp;
        target = target->next;
    }
}

```

delete
掉旧的

}

EECS 280 Fall 2017 Final 10/16
uniqname (REQUIRED)

Problem 4: Iterators and Functors (16 Points)

4a) (3 points) Consider an IntRange class that is created with `start` and `stop` int values, representing the integers in the range `start, start+1, ..., stop-1` (it includes `start` but not `stop`). It can then be used inside a *range-based for* loop to execute the loop body with an integer starting at `start` and increasing by one each time until reaching `stop`.

```
IntRange range(1, 5);
for (int i : range) {
    cout << i << " ";
} // will output "1 2 3 4 "
```

int i = 10
cout << i << ", "

Here is the code for IntRange, which is missing the public interface for its Iterator nested class.

```
class IntRange {
public:
    IntRange(int start_in, int stop_in) : start(start_in), stop(stop_in) {}
    class Iterator {
public:
    // Your public member functions here
private:
    Iterator(int i_in) : i(i_in) {}
    int i;
    friend class IntRange;
};
Iterator begin() const { return Iterator(start); }
Iterator end() const { return Iterator(stop); }
private:
    int start, stop;
};
```

Which Iterator member functions **MUST** you implement in order for the iterator to be used in the

range-based for loop above?

Must be defined to be used above?	Yes	No
Overloaded != operator for <code>IntRange::Iterator</code>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Overloaded prefix * operator for <code>IntRange::Iterator</code>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Custom implementations of the Big Three for <code>IntRange::Iterator</code>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

EECS 280 Fall 2017 Final 11/16

unique name (REQUIRED)

4b) (4 points) Write the following `filtered_copy` function, which copies items of type `T` from the given range into the provided `dest` vector if they satisfy the given Predicate. You may NOT use anything from the standard library other than vector member functions.

```
// REQUIRES: 'begin' and 'end' are valid iterators
// 'begin' is before or equal to 'end'
// IterType fulfills the standard iterator interface // Predicate
fulfills the standard predicate interface // MODIFIES: the input vector
'dest'
// EFFECTS: For each item from 'begin' to 'end',
    // appends that item to the given vector 'dest' // if Predicate 'pred'
    // returns true for that item // EXAMPLE: begin and end define a sequence
containing {1, 8, 3, 4} // and pred = IsEven, where IsEven(x) is true if x is
even // Appends the items 8 and 4 to an input vector<int> 'dest' template
<typename IterType, typename Predicate, typename T> void
filtered_copy(IterType begin, IterType end, Predicate pred, vector<T> &dest)
{
```

```
void filtered_copy(IterType begin, IterType end, Predicate pred,
    vector<T> &dest) {
    int count = 0;
    for (IterType it = begin; it != end; ++it){
        if (pred(*it)){
            dest.at(count) = *it;
            ++count;
        }
    }
}
or
void filtered_copy(IterType begin, IterType end, Predicate pred,
    vector<T> &dest) {
    for (;begin != end; ++begin){
        if (pred(*begin)){
            dest.push_back(*begin);
        }
    }
}
```

4c) (5 points) Complete the definition of the `IsPowerOfBase` functor below, which when created with a fixed integer base, can determine whether an input integer is a power of that base (using a positive integer exponent). For example, here the functor is defined with base 2:

```
IsPowerOfBase pow_of_2();
pow_of_2(8); // returns true since 8 is a power of 2 ( $2^3 = 8$ )
pow_of_2(5); // returns false since 5 is not a power of 2
```

You **MUST** use this helper function:

```
// REQUIRES: num > 0, base > 0
// EFFECTS: Returns true if 'num' is a power of 'base'
//           ( $num = base^{exp}$  for some integer  $exp > 0$ )
// EXAMPLE: is_power_of(16, 4) is true,
//           is_power_of(9, 2) is false
bool is_power_of(int num, int base);
```

You do not have to do any error checking in your code below.

```
class IsPowerOfBase {
public:
    IsPowerOfBase( int base_input );
    : base_Private( base_input );
    bool operator( )( int num_input ) const {
        return is_power_of( num_input, base_private );
    }
private:
    base_Private;
```

unique name (REQUIRED) _____

```
class IntRange {
public:
    IntRange(int start_in, int stop_in);
    class Iterator {
        public:
            // Public member functions as required by 4a)
    };
    Iterator begin() const;
    Iterator end() const;
};

template <typename IterType, typename Predicate, typename T>
void filtered_copy(IterType begin, IterType end, Predicate pred,
                  vector<T> &dest);
```

Public interfaces from 4a) and 4b)

4d) (4 points) In the box below, use the `filtered_copy` function to fill the `powers_of_three` vector with all the positive integers less than 1000 that are a power of 3. Your solution must only use `IntRange`, `IsPowerOfBase`, and `filtered_copy`. You may **NOT** use loops or recursion. You may assume that the member functions of `IntRange::Iterator` required by 4a) are implemented.

```
vector<int> powers_of_three;

vector<int> resorce_vector;
for (int i = 0; i < 1000; ++i){
    resorce_vector.push_back(i);
}

/*IntRange<int>::Iterator*/ IterType begin = resorce_vector.begin();
/*IntRange<int>::Iterator*/ IterType end = resorce_vector.end();
IsPowerOfBase ipob(3);
filtered_copy(begin, end, ipob, powers_of_three);
```

// Now `powers_of_three` will have { 3, 9, 27, 81, 243, 729 }

```
IntRange range(1,1000);
filtered_copy(range.begin(),range.end(),IsPowerOfBase(3),powers_of_three);
```

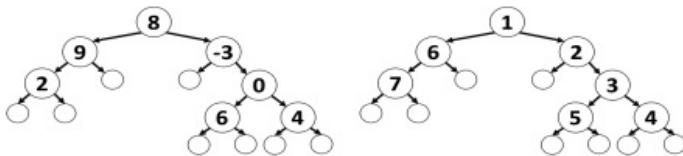
Problem 5: Recursion (14 Points)

For this question, work directly with the node struct to the right, representing a tree containing ints. You may NOT use any functions except those defined in this problem. Your functions MUST be recursive, and you may NOT use loops.

5a) (7 points) The call `same_structure(node1, node2)` determines whether or not `node1` and `node2` have the same structure, with the same height and nodes at the same positions in each tree. The following two trees have the same structure:

```
struct Node {  
    int datum;  
    Node *left;  
    Node *right;  
};  
// a null pointer  
// represents  
// an empty tree
```

Node struct



Implement `same_structure()` below.

```
// EFFECTS: Returns true if the nodes in 'node1' have the same  
// structure as those in 'node2'.
```

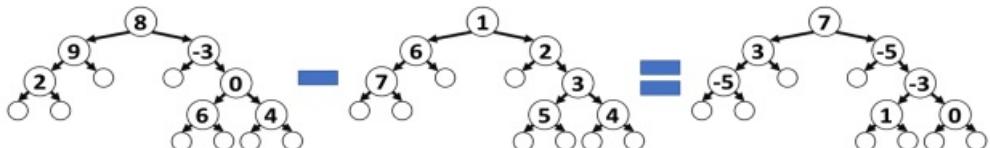
```
bool same_structure(const Node *node1, const Node *node2) {
```

```
bool same_structure (const Node *node1, const Node *node2){  
    if(!node1&&!node2){  
        return true;  
    }  
    if((!node1&&node2) || (node1&&!node2)) {  
        return false;  
    }  
    else{  
        return same_structure(node1->left, node2->left)&&same_structure(node1->right, node2->right);  
    }  
}
```

```
if (node1 and node2) {  
    return same_structure(node1->left,  
                         node2->left) and  
           same_structure(node1->right,  
                         node2->right);  
}  
return node1 == node2;
```

right right

5b) (7 points) Implement the `combine()` function template below. (You do not have to assert the REQUIRES clause.) Given two trees and a function object `func` that takes two ints and returns an int, `combine()` constructs a new tree that is the result of applying `func` to each pair of elements that are at the same position in the two original trees. For example, if `node1` is the tree on the left, `node2` is the middle tree, and `func` is a function object that subtracts two ints, then the result of `combine(node1, node2, func)` is the tree on the right:



```
// REQUIRES: same_structure(node1, node2)
// EFFECTS: Returns a node structure that has the same structure as
//           'node1' and 'node2', where the element at each position
//           is the result of applying 'func' to the elements at that
//           position in 'node1' and 'node2'.
template <typename Combiner>
Node * combine(const Node *node1, const Node *node2, Combiner func) {
```

```
    Node * combine (const Node *node1, const Node *node2, Combiner func) {
        if(node1) {
            Node* root = new Node;
            root->datum = func(node1->datum, node2->datum);
            root->left = combine(node1->left, node2->left);
            root->right = combine(node1->right, node2->right);
            return root;
        }
        else{
            return nullptr;
        }
    }
```

↙

```
if (not node1) {
    return nullptr;
}
return new Node {
    func(node1->datum, node2->datum),
    combine (node1->left, node2->left, func),
    combine (node1->right, node2->right, func)
};
```

}

EECS 280 Fall 2017 Final 16/16
